**Group Members:** 1. Anwoy Chatterjee (**Entry Number:** 2022AIB2675)

2. Pratik Vora (**Entry Number:** 2022AIB2687)

**Modified Greedy Local Search with sliding window and increased exploration:**

**State Representation:** The sentence (in string format) is used as the representation for the states.

**Intuition behind the algorithm:** We have implemented an algorithm which is based on the Greedy Hill-climbing search. But unlike the original algorithm which stops when all the successors of a state have higher cost than the state itself, our algorithm stops only when we have iterated the whole string (given as start state) once. This increases the exploration of search space by the algorithm, as else it, being a local search algorithm, explores a very limited section of search space and stops exploration much early. We have generated the successors for each state based on a window size, which is discussed below - using this window helps us to handle the trade-off between the extensive exploration of search space and the time taken by the algorithm.

We have taken a window size of w = 3. The window used here behaves as a sliding window.

**Generation of successors:** First we take all possible combinations of corrections of the w letters on which the window is currently placed. As we have taken w =3, there will be such (p\*q\*r) possible combinations, where ‘p’, ‘q’, ‘r’ are the number of possible corrections (including the letter itself) available for the first, second and third letter in the window respectively, based on the conf\_matrix given.

It is to be noted that the given conf\_matrix actually provides the list of all possible letters which can be extracted in place of the key of the dictionary conf\_matrix - from this given dictionary we build the dictionary “possiblity\_matrix” for which the value for a key lists the all possible choices for replacement of the key, and the value list also includes the key. The possibility\_matrix is a kind of transpose for the given conf\_matrix.

Among the (p\*q\*r) successor generated for a node, we then take the successor with minimum cost and expand it further.

**Branching factor:** As discussed above, as we have used a window size of w =3, the branching factor in our case is (p\*q\*r), where ‘p’, ‘q’, ‘r’ are the number of possible corrections (including the letter itself) available for the first, second and third letter in the window respectively, based on the possiblity\_matrix built by us, as mentioned above.

**Algorithm:**

1. For each iteration repeat the steps below; repeat for (L-w) iterations, where, L is the length of the input string (or start state), and w is the window size we are using, i.e. until the sliding window has traversed through the whole string once.
2. Initialize: current\_state = start\_state and best\_state = start\_state
3. Find all possible combinations of the three letters within the sliding window(based on the correction choices available for each word). Generate all possible successors of the current\_state by taking all such possible combinations by keeping letters outside the sliding window unchanged.
4. Update: current\_state = successor with minimum cost
5. If cost of current\_state < cost of best\_state, update: best\_state=current\_state